DTIC FILE COPY

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

2

AD-A227 064

to average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and ... ion. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions ... ctorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to ... t and Budget, Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final 27 June 1990 to 27 June 1991

4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report: ALSYS LIMITED, AlsyCOMP_042, Version 5.3, IBM 9370 Model 90 under AIX/370 Version 1.2 (Host & Target), 900627N1.11013

5. FUNDING NUMBERS

6. AUTHOR(S)
National Computing Centre Limited
Manchester, UNITED KINGDOM

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
National Computing Centre Limited
Oxford Road
Manchester MI 7ED
UNITED KINGDOM

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-VSR90502/900730

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

ALSYS LIMITED, AlsyCOMP_042, Version 5.3, Manchester, Englan, IBM 9370 Model 90 under AIX/370, Version 1.2 (Host & Target), ACVC 1.11.

14. SUBJECT TERMS Ada programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, Validation Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-STD-1815A, Ada Joint Program Office

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

NSN 7540-01-280-5500

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-18
299-01

CHAPTER 1

INTRODUCTION

This Validation Summary Report ~~(VSR)~~ describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability ~~(ACVC)~~. An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation-dependent but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

(KR)

AVF Control Number: AVF-VSR90502/69-900730

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #900627N1.11013
ALSYS LIMITED
AlsyCOMP_042 Version 5.3
IBM 9370 Model 90 under AIX/370 Version 1.2

Prepared by
Testing Services
The National Computing Centre Limited
Oxford Road
Manchester
M1 7ED
England

VSR Version 90-04-03

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on **27 June 1990.**

Compiler Name and Version: **AlsyCOMP_042 Version 5.3**

Host Computer System: **IBM 9370 Model 90 under AIX/370 Version 1.2**

Target Computer System: **IBM 9370 Model 90 under AIX/370 Version 1.2**

A more detailed description of this Ada implementation is found in section 3.1 of this report. As a result of this validation effort, Validation Certificate [redacted] is awarded to **Alsys Limited.** This certificate expires on **June 1 1992.**

This report has been reviewed and is approved.

Jane Pink
Testing Services Manager
The National Computing Centre Limited
Oxford Road
Manchester
M1 7ED
England

Ada Validation Organization
Director, Computer & Software
Engineering Division
Institute for Defense Analyses
Alexandria
VA 22311

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington
DC 20301

The following declaration of conformance was supplied by the customer.

DECLARATION OF CONFORMANCE

Customer: **ALSYS LIMITED**

Ada Validation Facility: **The National Computing Centre Limited**
Oxford Road
Manchester
M1 7ED
United Kingdom

ACVC Version: 1.11

Ada Implementation:

Ada Compiler Name: **AlsyCOMP_042**

Version: **5.3**

Host Computer System: **IBM 9370 Model 90 under AIX/370. Version 1.2.**

Target Computer System: **IBM 9370 Model 90 under AIX/370. Version 1.2**

Customer's Declaration

I, the undersigned, representing **Alsys Limited**, declare that **Alsys Limited** has no knowledge of deliberate deviations from the Ada Language Standard **ANSI/MIL-STD-1815A** in the implementation listed in this declaration.

MARTYN JORDAN
MARKETING DIRECTOR

Date: 27/6/90

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro89] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro89]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

> **National Technical Information Service**
> **5285 Port Royal Road**
> **Springfield**
> **VA 22161**

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

> **Ada Validation Organization**
> **Institute for Defense Analyses**
> **1801 North Beauregard Street**
> **Alexandria**
> **VA 22311**

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987

[Pro89] Ada Compiler Validation Procedures, Version 2.0, Ada Joint Program Office, May 1989.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behaviour is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 3.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability Report	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada Certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfilment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.

Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro89].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 71 tests had been withdrawn by the Ada Validation Organization (AVO) at the time of validation testing. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 90-05-18.

E28005C	B38006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
B83022B	B83022H	B83025B	B83025D	B83026B	C83026A
C83041A	C97116A	C98003B	BA2011A	CB7001A	CB7001B
CB7004A	CC1223A	BC1226A	CC1226B	BC3009B	AD1B08A
BD2A02A	CD2A21E	CD2A23E	CD2A32A	CD2A41A	CD2A41E
CD2A87A	CD2B15C	BD3006A	CD4022A	CD4022D	CD4024B
CD4024C	CD4024D	CD4031A	CD4051D	CD5111A	CD7004C
ED7005D	CD7005E	AD7006A	CD7006E	AD7201A	AD7201E
CD7204B	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CD2119B	CE2205B	CE2405A	CE3111C	CE3118A
CE3411B	CE3412B	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. The inapplicability criteria for some tests are explained in documents issued by ISO and the AJPO known as Ada Issues and commonly referenced in the format AI-dddd. For this implementation, the following tests were inapplicable for the reasons indicated; references to Ada Issues are included as appropriate.

The following 159 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113O..Y (11 tests)	C35705O..Y (11 tests)
C35706O..Y (11 tests)	C35707O..Y (11 tests)
C35708O..Y (11 tests)	C35802O..Z (12 tests)
C45241O..Y (11 tests)	C45321O..Y (11 tests)
C45421O..Y (11 tests)	C45521O..Z (12 tests)
C45524O..Z (12 tests)	C45621O..Z (12 tests)
C45641O..Y (11 tests)	C46012O..Z (12 tests)

The following 21 tests check for the predefined type LONG_INTEGER:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

C45531M..P (4 tests and C45532M..P (4 tests)) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater.

C45536A, C46013B, C46031B, C46033B and C46034B contain 'SMALL representation clauses which are not powers of two or ten.

C45624A checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 5, For this implementation, MACHINE_OVERFLOWS is TRUE.

C45624B checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 6. For this implementation, MACHINE_OVERFLOWS is TRUE.

C86001F checks that package SYSTEM is used by package TEXT_IO.

B86001Y checks for a predefined fixed-point type other than DURATION.

C96005B checks for values of type DURATION'BASE that are outside the range of DURATION. There are no such values for this implementation.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A53A checks operations of a fixed-point type for which a length clause specified a power-of-ten type'small; the AVO rules that, under ACVC 1.11, support of decimal smalls may be omitted.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

CE2203A and CE2403A check that the capacity of an external file is limited.

CE3202A checks that unique names are given to the Standard Input and Standard Output.

EE2401D and EE2401G checks that instantiations for DIRECT_IO for unconstrained types are supported. This implementation requires a FORM parameter to be used to specify the maximum runtime size of any value of the type for which IO is to be performed.

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

The tests listed in the following table are not applicable because the given file operations are not supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2105A	CREATE	IN_FILE	SEQUENTIAL_IO
CE2105B	CREATE	IN_FILE	DIRECT_IO
CE2401H	CREATE	IN_FILE	DIRECT_IO
CE3109A	CREATE	IN_FILE	TEXT_IO

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 18 tests. Although 19 files are affected, only 18 tests required modification.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B23004A	B24007A	B24009A	B28003A	B32202A	B32202B
B32202C	B37004A	B45102A	B61012A	B91004A	B95069A
B95069B	B97103E	BA1101B2	BA1101B4	BC2001D	BC3009C

BA2001E - Errors were detected, as stated in the test listings, during the bind rather than at compilation; the AVO ruled that this behaviour is acceptable.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

Testing of the AlsyCOMP_042 Version 5.3 compiler using ACVC version 1.11 was performed on a configuration described below:

Host computer:	IBM 9370 model 90
Host Operating System:	AIX/370 Version 2.1

Target computer and operating system are as host.

Compiler:	AlsyCOMP_042 Version 5.3
Pre-linker:	AlsyCOMP_042 Version 5.3
Linker:	AIX/370 linker, Id.
Runtime System:	AlsyCOMP_042 Version 5.3

For a point of contact for technical information about this Ada implementation system, see:

Simon FitzMaurice
Alsys Limited
Newtown Road
Henley-on-Thames
Oxon
RG9 1EN

For a point of contact for sales information about this Ada implementation system, see:

John Stewart
Alsys Limited
Newtown Road
Henley-on-Thames
RG9 1EN

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 TEST EXECUTION

Version 1.11 of the ACVC comprises **4170** tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that **241** tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for **159** executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A **magnetic tape** containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the **magnetic tape** were not loaded directly onto the host computer, but loaded onto a disc attached to a Sun Workstation.

The files were then accessed from the host computer using network file access.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options.

Test output, compiler and linker listings, and job logs were captured on **Magnetic Tape** and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The following macro parameters are defined in terms of the value of V of $MAX_IN_LEN which is the maximum input line length permitted for the tested implementation. For these parameters, Ada string expressions are given rather than the macro values themselves.

Macro Parameter	Macro Value
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

Macro Parameter	Macro Value
$MAX_IN_LEN	255
$ACC_SIZE	32
$ALIGNMENT	04
$COUNT_LAST	2147483647
$DEFAULT_MEM_SIZE	4294967296
$DEFAULT_STOR_UNIT	08
$DEFAULT_SYS_NAME	S370
$DELTA_DOC	2:1.0:E-31
$ENTRY_ADDRESS	SYSTEM.NULL_ADDRESS
$ENTRY_ADDRESS1	SYSTEM.NULL_ADDRESS
$ENTRY_ADDRESS2	SYSTEM.NULL_ADDRESS
$FIELD_LAST	255
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100000.0
GREATER_THAN_DURATION_BASE_LAST	10000000.0
$GREATER_THAN_FLOAT_BASE_LAST	1.0E+80
$GREATER_THAN_FLOAT_SAFE_LARGE	2.0**252*(1.0-2.0**(-54))

$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.0E+40
$HIGH_PRIORITY	10
$ILLEGAL_EXTERNAL_FILE_NAME1	/ /
$ILLEGAL_EXTERNAL_FILE_NAME2	/ . /
$INAPPROPRIATE_LINE_LENGTH	-01
$INAPPROPRIATE_PAGE_LENGTH	-01
$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.ADA")
$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006E1.ADA")
$INTEGER_FIRST	-2147483648
$INTEGER_LAST	2147483647
$INTEGER_LAST_PLUS_1	2147483648
$INTERFACE_LANGUAGE	ASSEMBLER
$LESS_THAN_DURATION	-100000.0
$LESS_THAN_DURATION_BASE_FIRST	-10000000.0
$LINE_TERMINATOR	ASCII.LF
$LOW_PRIORITY	01
$MACHINE_CODE_STATEMENT	NULL;
$MACHINE_CODE_TYPE	NO_SUCH_TYPE
$MANTISSA_DOC	31
$MAX_DIGITS	18
$MAX_INT	2147483647
$MAX_INT_PLUS_1	2_147_483_648

$MIN_INT	-2147483648
$NAME	SHORT_SHORT_INTEGER
$NAME_LIST	I80X86, I80386, MC680X0, S370, TRANSPUTER, VAX
$NAME_SPECIFICATION1	/aix370/tmp/X2120A
$NAME_SPECIFICATION2	/aix370/tmp/X2120B
$NAME_SPECIFICATION3	/aix370/tmp/X3119A
$NEG_BASED_INT	16#FFFFFFFF#
$NEW_MEM_SIZE	00
$NEW_STOR_UNIT	00
$NEW_SYS_NAME	I80X86
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	NEW INTEGER;
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	102400
$TICK	0.01
$VARIABLE_ADDRESS	V_ADDRESS
$VARIABLE_ADDRESS1	V_ADDRESS1
$VARIABLE_ADDRESS2	V_ADDRESS2
$YOUR_PRAGMA	NO_SUCH_PRAGMA

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report.

Standard Options

CALLS=INLINED	Allow inline insertion of code for subprograms.
OBJECT=NONE	Do not perform peephole optimisations.
NOWARN	Do not generate warning messages.
REDUCTION=EXTENSIVE	High level optimization is performed.
EXPRESSIONS=EXTENSIVE	Low level optimization is performed.

Options for C tests (no compilation listing options)

OUT=NONE	Do not produce any compilation listing at all.

Options for B and NA tests (compilation listing options)

TEXT	Include the full text of the compilation with embedded error messages in the generated compilation listing.
SHOW=NONE	Suppress banner header on each listing page. Do not include error summary at end of listing.
ERRORS=999	Allow 999 errors before terminating the compilation.
NODETAIL	Do not include extra detail in error messages generated by the compiler.
FILE_WIDTH=79	Listing file has 79 characters per line (lines folded as appropriate).
FILE_LENGTH=32767	Listing file has 32767 lines per page (in effect, an unpaginated listing file).

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

Binder options used:

For L Tests:	NOHIS	Do not include in-bound image data for dynamic traceback of exceptions.
	NOWARN	Do not generate warning messages.
	FILE_WIDTH=79	Listing file line length=79
For others:	NOHIS	Do not include in bound image data for dynamic traceback of exceptions.
	OUT=NONE	Do not produce a listing.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

        type SHORT_SHORT_INTEGER is range
            -128 .. 127;
        type SHORT_INTEGER is range
            -32_768 .. 32_767;
        type INTEGER is range
            -2_147_483_648 .. 2_147_483_647;
        type SHORT_FLOAT is digits 6 range
            -16#0.FFFF_FF#E+63 ..
             16#0.FFFF-FF#E+63;
        type FLOAT is digits 15 range
            -16#0.FFFF_FFFF_FFFF_FF#E+63 ..
             16#0.FFFF_FFFF_FFFF_FF#E+63;
        type LONG_FLOAT is digits 18 range
            -16#0.FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF#E+63 ..
             16#0.FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF#E+63;
        type DURATION is delta 2#0.000_000_000_000_01# range
            -131_072.0 .. 131_071.0;

end STANDARD;
```

Alsys IBM 370 Ada Compiler

APPENDIX F

for AIX

Implementation - Dependent Characteristics

Version 5.3

Alsys S.A.
29, Avenue Lucien-René Duchesne
78170 La Celle St. Cloud, France

Alsys Inc.
67 South Bedford Street
Burlington, MA 01803-5152, U.S.A.

Alsys Ltd.
Partridge House, Newtown Road
Henley-on-Thames,
Oxfordshire RG9 1EN, U.K.

Printed: May 24, 1990

Alsys reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should in all cases consult Alsys to determine whether such changes have been made.

PREFACE

This *Alsys IBM 370 Ada Compiler Appendix F for AIX* is for programmers, software engineers, project managers, educators and students who want to develop an Ada program for any IBM System/370 processor that runs AIX.

This appendix is a required part of the *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD 1815A, January 1983 (throughout this appendix, citations in square brackets refer to this manual). It assumes that the user is already familiar with the AIX operating system, and has access to the following IBM documents:

Commands Reference, Volume 1, SC23-2025

Commands Reference, Volume 2, SC23-2184

Programming Tools and Interfaces, SC23-2029

Using the Operating System, SC23-2024

IBM System 370 Principles of Operation, GA22-7000

IBM System 370 System Summary, GA22-7001

TABLE OF CONTENTS

APPENDIX F

Implementation-Dependent Characteristics

This appendix summarizes the implementation-dependent characteristics of the Alsys IBM 370 Ada Compiler for AIX. This document should be considered as the Appendix F to the Reference Manual for the Ada Programming Language ANSI MIL-STD 1815A, January 1983, as appropriate to the Alsys Ada implementation for the IBM 370 under AIX.

Sections 1 to 8 of this appendix correspond to the various items of information required in Appendix F [F]*; sections 9 and 10 provide other information relevant to the Alsys implementation. The contents of these sections is described below:

1. The form, allowed places, and effect of every implementation-dependent pragma.

2. The name and type of every implementation-dependent attribute.

3. The specification of the package SYSTEM [13.7].

4. The list of all restrictions on representation clauses [13.1].

5. The conventions used for any implementation-generated names denoting implementation-dependent components [13.4].

6. The interpretation of expressions that appear in address clauses.

7. Any restrictions on unchecked conversions [13.10.2].

8. Any implementation-dependent characteristics of the input-output packages [14].

9. Characteristics of numeric types.

---

* Throughout this manual, citations in square brackets refer to the *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, January 1983.

10. Other implementation-dependent characteristics.

Throughout this appendix, the name *Ada Run-Time Executive* refers to the run-time library routines provided for all Ada programs. These routines implement the Ada heap, exceptions, tasking control, I/O, and other utility functions.

1 Implementation-Dependent Pragmas

1.1 INLINE

Pragma INLINE is fully supported, except for the fact that it is not possible to inline a function call in a declarative part.

1.2 INTERFACE

Ada programs can interface to subprograms written in C or other languages through the use of the predefined pragma INTERFACE [13.9] and the implementation-defined pragma INTERFACE_NAME.

Pragma INTERFACE specifies the name of an interfaced subprogram and the name of the programming language for which calling and parameter passing conventions will be generated. Pragma INTERFACE takes the form specified in the *Reference Manual*:

pragma INTERFACE (*language_name*, *subprogram_name*);

where:

- *language_name* is the name of the other language whose calling and parameter passing conventions are to be used.
- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.

The only language names currently accepted by pragma INTERFACE are C and ASSEMBLER.

The language name used in the pragma INTERFACE does not necessarily correspond to the language used to write the interfaced subprogram. It is used only to tell the Compiler how to generate subprogram calls, that is, which calling conventions and parameter passing techniques to use.

The language name C is used to refer to the standard IBM AIX/370 C calling and parameter passing conventions. The programmer can use the language name C to interface Ada subprograms with subroutines written in any language that follows the standard IBM AIX/370 C calling conventions.

The language name ASSEMBLER provides the same calling and parameter passing conventions as the language name C, and only differs in the handling of the external name, as described below.

1.3 INTERFACE_NAME

Pragma INTERFACE_NAME associates the name of an interfaced subprogram, as declared in Ada, with its name in the language of origin. If pragma INTERFACE_NAME is not used, then the two names are assumed to be identical.

This pragma takes the form:

> **pragma** INTERFACE_NAME (*subprogram_name*, *string_literal*);

where:

- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.
- *string_literal* is the name by which the interfaced subprogram is referred to at link-time.

The use of INTERFACE_NAME is optional, and is not needed if a subprogram has the same name in Ada as in the language of origin. It is necessary, for example, if the name of the subprogram in its original language contains characters that are not permitted in Ada identifiers. Ada identifiers can contain only letters, digits and underscores, whereas the IBM AIX/370 linkage editor (ld) allows external names to contain other characters, e.g. the plus or minus sign. These characters can be specified in the *string_literal* argument of the pragma INTERFACE_NAME.

The pragma INTERFACE_NAME is allowed at the same places of an Ada program as the pragma INTERFACE [13.9]. However, the pragma INTERFACE_NAME must always occur after the pragma INTERFACE declaration for the interfaced subprogram.

There is no limit to the length of the *string_literal* nor any restriction on the characters of which is it composed. The user must be aware however, that some tools from other

vendors may not fully support the standard object file format and may restrict the length or character content of symbols.

For a subprogram interfaced using the language name ASSEMBLER, the external name of the subprogram passed through to the AIX object file is the string literal used in the pragma INTERFACE_NAME, with case preserved. If a pragma INTERFACE_NAME is not used, the Ada name of the subprogram is passed through to the object file, in lower case.

The external name passed through to the object file for a subprogram interfaced using the language name C is formed in the same way as for a subprogram interfaced using ASSEMBLER, with the addition of a leading underscore character. This follows the same conventions as used by the AIX/370 C compiler and assembler.

The *Runtime Executive* contains several external identifiers. The majority of these identifiers begin with the string "ALSY" or the string "alsy". Accordingly, external names of this form should be avoided by the user.

Example

```
package SAMPLE_DATA is
    function SAMPLE_DEVICE  (X : INTEGER) return INTEGER;
    function PROCESS_SAMPLE (X : INTEGER) return INTEGER;
    private
        pragma INTERFACE (C, SAMPLE_DEVICE);
        pragma INTERFACE (C, PROCESS_SAMPLE);
        pragma INTERFACE_NAME (PROCESS_SAMPLE, "PSAMPLE");
end SAMPLE_DATA;
```

1.4 INDENT

This pragma is only used with the Alsys Reformatter (*AdaReformat*); this tool offers the functionalities of a source reformatter in an Ada environment.

The pragma is placed in the source file and interpreted by the Reformatter.

pragma INDENT(OFF)

The Reformatter does not modify the source lines after the OFF pragma INDENT.

pragma INDENT(ON)

The Reformatter resumes its action after the ON pragma INDENT. Therefore any source lines that are bracketed by the OFF and ON pragma INDENTs are not modified by the Alsys Reformatter.

1.5 Other Pragmas

Pragmas IMPROVE and PACK are discussed in detail in the section on representation clauses (Chapter 4).

Pragma PRIORITY is accepted with the range of priorities running from 1 to 10 (see the definition of the predefined package SYSTEM in Chapter 3). The undefined priority (no pragma PRIORITY) is treated as though it were less than any defined priority value.

In addition to pragma SUPPRESS, it is possible to suppress checks in a given compilation by the use of the Compiler option CHECKS.

The following language defined pragmas have no effect.

CONTROLLED
MEMORY_SIZE
OPTIMIZE
STORAGE_UNIT
SYSTEM_NAME

Note that all access types are implemented by default as controlled collections as described in [4.8] (see section 10.1).

2 Implementation-Dependent Attributes

In addition to the Representation Attributes of [13.7.2] and [13.7.3], the four attributes listed in section 5 (Conventions for Implementation-Generated Names), for use in record representation clauses, and the attributes described below are provided:

T'DESCRIPTOR_SIZE	For a prefix T that denotes a type or subtype, this attribute yields the size (in bits) required to hold a descriptor for an object of the type T, allocated on the heap or written to a file. If T is constrained, T'DESCRIPTOR_SIZE will yield the value 0.

T'IS_ARRAY

For a prefix T that denotes a type or subtype, this attribute yields the value TRUE if T denotes an array type or an array subtype; otherwise, it yields the value FALSE.

Limitations on the use of the attribute ADDRESS

The attribute ADDRESS is implemented for all prefixes that have meaningful addresses. The following entities do not have meaningful addresses. The attribute ADDRESS will deliver the value SYSTEM.NULL_ADDRESS if applied to such prefixes and a compilation warning will be issued.

- A constant or named number that is implemented as an immediate value (i.e. does not have any space allocated for it).
- A package specification that is not a library unit.
- A package body that is not a library unit or subunit.

3 Specification of the Package SYSTEM

```
package SYSTEM is

   type NAME is (I80X86,
                 I80386,
                 MC680X0,
                 S370,
                 TRANSPUTER,
                 VAX);

   SYSTEM_NAME  : constant NAME := S370;

   STORAGE_UNIT : constant := 8;
   MAX_INT      : constant := 2**31 - 1;
   MIN_INT      : constant := - (2**31);
   MAX_MANTISSA : constant := 31;
   FINE_DELTA   : constant := 2#1.0#E-31;
   MAX_DIGITS   : constant := 18;
   MEMORY_SIZE  : constant := 2**32;
   TICK         : constant := 0.01;

   subtype PRIORITY is INTEGER range 1 .. 10;

   type ADDRESS is private;
   NULL_ADDRESS : constant ADDRESS;
```

```
    function VALUE (LEFT : in STRING) return ADDRESS;

    subtype ADDRESS_STRING is STRING(1..8);

    function IMAGE (LEFT : in ADDRESS) return ADDRESS_STRING;


    type OFFSET is range -(2**31) .. 2**31-1;
    -- This type is used to measure a number of storage units (bytes).

    function SAME_SEGMENT (LEFT, RIGHT : in ADDRESS) return BOOLEAN;

    ADDRESS_ERROR : exception;


    function "+" (LEFT : in ADDRESS; RIGHT : in OFFSET)  return ADDRESS;
    function "+" (LEFT : in OFFSET;  RIGHT : in ADDRESS) return ADDRESS;
    function "-" (LEFT : in ADDRESS; RIGHT : in OFFSET)  return ADDRESS;

    function "-" (LEFT : in ADDRESS; RIGHT : in ADDRESS) return OFFSET;

    function "<=" (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
    function "<"  (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
    function ">=" (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
    function ">"  (LEFT, RIGHT : in ADDRESS) return BOOLEAN;

    function "mod" (LEFT : in ADDRESS; RIGHT : in POSITIVE) return NATURAL;


    type ROUND_DIRECTION is (DOWN, UP);

    function ROUND (VALUE     : in ADDRESS;
                    DIRECTION : in ROUND_DIRECTION;
                    MODULUS   : in POSITIVE) return ADDRESS;


    generic
       type TARGET is private;
    function FETCH_FROM_ADDRESS (A : in ADDRESS) return TARGET;
    generic
       type TARGET is private;
    procedure ASSIGN_TO_ADDRESS (A : in ADDRESS; T : in TARGET);
    -- These routines are provided to perform READ/WRITE operations in memory.


    type OBJECT_LENGTH is range 0 .. 2**31 -1;
    -- This type is used to designate the size of an object in storage units.

    procedure MOVE (TO     : in ADDRESS;
                    FROM   : in ADDRESS;
                    LENGTH : in OBJECT_LENGTH);

end SYSTEM;
```

The function VALUE may be used to convert a string into an address. The string is a sequence of up to eight hexadecimal characters (digits or letters in upper or lower case in the range A..F) representing a virtual address. The exception CONSTRAINT_ERROR is raised if the string does not have the proper syntax.

The function IMAGE may be used to convert an address to a string which is a sequence of exactly eight hexadecimal digits.

The function SAME_SEGMENT always returns TRUE and the exception ADDRESS_ERROR is never raised as the 370 is a non segmented architecture.

The functions "+" and "-" with an ADDRESS and an OFFSET parameter provide support to perform address computations. The OFFSET parameter is added to, or subtracted from the address. The exception CONSTRAINT_ERROR can be raised by these functions.

The function "-" with the two ADDRESS parameters may be used to return the distance between the specified addresses.

The functions "<=", "<", ">=" and ">" may be used to perform a comparison on the specified addresses. The comparison is unsigned.

The function "mod" may be used to return the offset of LEFT address relative to the memory block immediately below it starting at a multiple of RIGHT storage units.

The function ROUND may be used to return the specified address rounded to a specific value in a particular direction.

The generic function FETCH_FROM_ADDRESS may be used to read data objects from given addresses in store. The generic function ASSIGN_TO_ADDRESS may be used to write data objects to given addresses in store. These routines may not be instantiated with unconstrained types.

The procedure MOVE may be used to copy LENGTH storage units starting at the address FROM to the address TO. The source and destination locations may overlap.

4 Restrictions on Representation Clauses

This section explains how objects are represented and allocated by the Alsys IBM 370 Ada Compiler and how it is possible to control this using representation clauses.

The representation of an object is closely connected with its type. For this reason this section addresses successively the representation of enumeration, integer, floating point, fixed point, access, task, array and record types. For each class of type the representation of the corresponding objects is described.

Except in the case of array and record types, the description of each class of type is independent of the others. To understand the representation of an array type it is necessary to understand first the representation of its components. The same rule applies to a record type.

Apart from implementation defined pragmas, Ada provides three means to control the size of objects:

- a (predefined) pragma PACK, when the object is an array, an array component, a record or a record component
- a record representation clause, when the object is a record or a record component
- a size specification, in any case.

For each class of types the effect of a size specification is described. Interaction between size specifications, packing and record representation clauses is described under array and record types.

Size representation clauses on types derived from private types are not supported when the derived type is declared outside the private part of the defining package.

4.1 Enumeration Types

Internal codes of enumeration literals

When no enumeration representation clause applies to an enumeration type, the internal code associated with an enumeration literal is the position number of the enumeration literal. Then, for an enumeration type with n elements, the internal codes are the integers 0, 1, 2, ... , n-1.

An enumeration representation clause can be provided to specify the value of each internal code as described in [13.3]. The Alsys Compiler fully implements enumeration representation clauses.

As internal codes must be machine integers the internal codes provided by an enumeration representation clause must be in the range $-2^{31} .. 2^{31}-1$.

Encoding of enumeration values

An enumeration value is always represented by its internal code in the program generated by the Compiler.

Enumeration subtypes

Minimum size: The minimum size of an enumeration subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form.

For a static subtype, if it has a null range its minimum size is 1. Otherwise, if m and M are the values of the internal codes associated with the first and last enumeration values of the subtype, then its minimum size L is determined as follows. For $m >= 0$, L is the smallest positive integer such that $M <= 2^{L}-1$. For $m < 0$, L is the smallest positive integer such that $-2^{L-1} <= m$ and $M <= 2^{L-1}-1$.

For example:

```
type COLOR is (GREEN, BLACK, WHITE, RED, BLUE, YELLOW);
-- The minimum size of COLOR is 3 bits.

subtype BLACK_AND_WHITE is COLOR range BLACK .. WHITE;
-- The minimum size of BLACK_AND_WHITE is 2 bits.

subtype BLACK_OR_WHITE is BLACK_AND_WHITE range X .. X;
-- Assuming that X is not static, the minimum size of BLACK_OR_WHITE is
-- 2 bits (the same as the minimum size of the static type mark
-- BLACK_AND_WHITE).
```

Size: When no size specification is applied to an enumeration type or first named subtype, the objects of that type or first named subtype are represented as signed integers if the internal code associated with the first enumeration value is negative, and as unsigned integers otherwise. The machine provides 8, 16 and 32 bit integers, and the Compiler selects automatically the smallest machine integer which can hold each of the internal codes of the enumeration type (or subtype). The size of the enumeration type and of any of its subtypes is thus 8, 16 or 32 bits.

When a size specification is applied to an enumeration type, this enumeration type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies.

For example:

```
type EXTENDED is
 ( -- The usual American ASCII characters.
       NUL,   SOH,   STX,   ETX,   EOT,   ENQ,   ACK,   BEL,
       BS,    HT,    LF,    VT,    FF,    CR,    SO,    SI,
       DLE,   DC1,   DC2,   DC3,   DC4,   NAK,   SYN,   ETB,
       CAN,   EM,    SUB,   ESC,   FS,    GS,    RS,    US,
       ' ',   '!',   '"',   '#',   '$',   '%',   '&',   ''',
       '(',   ')',   '*',   '+',   ',',   '-',   '.',   '/',
       '0',   '1',   '2',   '3',   '4',   '5',   '6',   '7',
       '8',   '9',   ':',   ';',   '<',   '=',   '>',   '?',
       '@',   'A',   'B',   'C',   'D',   'E',   'F',   'G',
       'H',   'I',   'J',   'K',   'L',   'M',   'N',   'O',
       'P',   'Q',   'R',   'S',   'T',   'U',   'V',   'W',
       'X',   'Y',   'Z',   '[',   '\',   ']',   '^',   '_',
       '`',   'a',   'b',   'c',   'd',   'e',   'f',   'g',
       'h',   'i',   'j',   'k',   'l',   'm',   'n',   'o',
       'p',   'q',   'r',   's',   't',   'u',   'v',   'w',
       'x',   'y',   'z',   '{',   '|',   '}',   '~',   DEL,
       -- Extended characters
       LEFT_ARROW,
       RIGHT_ARROW,
       UPPER_ARROW,
       LOWER_ARROW,
       UPPER_LEFT_CORNER,
       UPPER_RIGHT_CORNER,
       LOWER_RIGHT_CORNER,
       LOWER_LEFT_CORNER,
       ...);

for EXTENDED'SIZE use 8;
-- The size of type EXTENDED will be one byte. Its objects will be represented
-- as unsigned 8 bit integers.
```

The Alsys Compiler fully implements size specifications. Nevertheless, as enumeration values are coded using integers, the specified length cannot be greater than 32 bits.

Object size: Provided its size is not constrained by a record component clause or a pragma PACK, an object of an enumeration subtype has the same size as its subtype.

Alignment: An enumeration subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, halfword aligned if the size of the subtype is less than or equal to 16 bits and word aligned otherwise.

Object address: Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an enumeration subtype is a multiple of the alignment of the corresponding subtype.

4.2 Integer Types

Predefined integer types

There are three predefined integer types in the Alsys implementation for IBM 370 machines:

type SHORT_SHORT_INTEGER	**is range** -2**07 .. 2**07-1;
type SHORT_INTEGER	**is range** -2**15 .. 2**15-1;
type INTEGER	**is range** -2**31 .. 2**31-1;

Selection of the parent of an integer type

An integer type declared by a declaration of the form:

type T **is range** L .. R;

is implicitly derived from either the SHORT_INTEGER or INTEGER predefined integer type. The Compiler automatically selects the predefined integer type whose range is the shortest that contains the values L to R inclusive. Note that the SHORT_SHORT_INTEGER representation is never automatically selected by the Compiler.

Encoding of integer values

Binary code is used to represent integer values, using a conventional two's complement representation.

Integer subtypes

Minimum size: The minimum size of an integer subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form (that is to say, in an unbiased form which includes a sign bit only if the range of the subtype includes negative values).

For a static subtype, if it has a null range its minimum size is 1. Otherwise, if m and M are the lower and upper bounds of the subtype, then its minimum size L is determined as follows. For m >= 0, L is the smallest positive integer such that $M <= 2^{L}-1$. For m < 0, L is the smallest positive integer such that $-2^{L-1} <= m$ and $M <= 2^{L-1}-1$.

For example:

subtype S **is** INTEGER **range** 0 .. 7;
-- The minimum size of S is 3 bits.

subtype D **is** S **range** X .. Y;
-- Assuming that X and Y are not static, the minimum size of
-- D is 3 bits (the same as the minimum size of the static type mark S).

Size: The sizes of the predefined integer types SHORT_SHORT_INTEGER, SHORT_INTEGER and INTEGER are respectively 8, 16 and 32 bits.

When no size specification is applied to an integer type or to its first named subtype (if any), its size and the size of any of its subtypes is the size of the predefined type from which it derives, directly or indirectly.

For example:

type S **is range** 80 .. 100;
-- S is derived from SHORT_INTEGER, its size is 16 bits.

type J **is range** 0 .. 65535;
-- J is derived from INTEGER, its size is 32 bits.

type N **is new** J **range** 80 .. 100;
-- N is indirectly derived from INTEGER, its size is 32 bits.

When a size specification is applied to an integer type, this integer type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies.

For example:

type S **is range** 80 .. 100;
for S'SIZE **use** 32;
-- S is derived from SHORT_INTEGER, but its size is 32 bits
-- because of the size specification.

type J **is range** 0 .. 255;
for J'SIZE **use** 8;
-- J is derived from SHORT_INTEGER, but its size is 8 bits because
-- of the size specification.

type N **is new** J **range** 80 .. 100;
-- N is indirectly derived from SHORT_INTEGER, but its size is 8 bits
-- because N inherits the size specification of J.

The Alsys Compiler implements size specifications. Nevertheless, as integers are implemented using machine integers, the specified length cannot be greater than 32 bits.

Object size: Provided its size is not constrained by a record component clause or a pragma PACK, an object of an integer subtype has the same size as its subtype.

Alignment: An integer subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, halfword aligned if the size of the subtype is less than or equal to 16 bits and word aligned otherwise.

Object address: Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an integer subtype is a multiple of the alignment of the corresponding subtype.

4.3 Floating Point Types

Predefined floating point types

There are three predefined floating point types in the Alsys implementation for IBM 370 machines:

> **type** SHORT_FLOAT **is**
> **digits** 6 **range** -2.0**252*(1.0-2.0**-24) .. 2.0**252*(1.0-2.0**-24);
>
> **type** FLOAT **is**
> **digits** 15 **range** -2.0**252*(1.0-2.0**-56) .. 2.0**252*(1.0-2.0**-56);
>
> **type** LONG_FLOAT **is**
> **digits** 18 **range** -2.0**252*(1.0-2.0**-112) .. 2.0**252*(1.0-2.0**-112);

Selection of the parent of a floating point type

A floating point type declared by a declaration of the form:

> **type** T **is digits** D [**range** L .. R];

is implicitly derived from a predefined floating point type. The Compiler automatically selects the smallest predefined floating point type whose number of digits is greater than or equal to D and which contains the values L and R.

Encoding of floating point values

In the program generated by the Compiler, floating point values are represented using the IBM 370 data formats for single precision, double precision and extended precision floating point values as appropriate.

Values of the predefined type SHORT_FLOAT are represented using the single precision format, values of the predefined type FLOAT are represented using the double precision format and values of the predefined type LONG_FLOAT are represented using the extended precision format. The values of any other floating point type are represented in

the same way as the values of the predefined type from which it derives, directly or indirectly.

Floating point subtypes

Minimum size: The minimum size of a floating point subtype is 32 bits if its base type is SHORT_FLOAT or a type derived from SHORT_FLOAT, 64 bits if its base type is FLOAT or a type derived from FLOAT and 128 bits if its base type is LONG_FLOAT or a type derived from LONG_FLOAT.

Size: The sizes of the predefined floating point types SHORT_FLOAT, FLOAT and LONG_FLOAT are respectively 32, 64 and 128 bits.

The size of a floating point type and the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly.

The only size that can be specified for a floating point type or first named subtype using a size specification is its usual size (32, 64 or 128 bits).

Object size: An object of a floating point subtype has the same size as its subtype.

Alignment: A floating point subtype is word aligned if its size is 32 bits and double word aligned otherwise.

Object address: Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a floating point subtype is a multiple of the alignment of the corresponding subtype.

4.4 Fixed Point Types

Small of a fixed point type

If no specification of small applies to a fixed point type, then the value of small is determined by the value of delta as defined by [3.5.9].

A specification of small can be used to impose a value of small. The value of small is required to be a power of two.

Predefined fixed point types

To implement fixed point types, the Alsys Compiler for IBM 370 machines uses a set of anonymous predefined types of the form:

type FIXED **is delta** D **range** (-2**15)*S .. (2**15-1)*S;
for FIXED'SMALL **use** S;

type LONG_FIXED **is delta** D **range** (-2**31)*S .. (2**31-1)*S;
for LONG_FIXED'SMALL **use** S;

where D is any real value and S any power of two less than or equal to D.

Selection of the parent of a fixed point type

A fixed point type declared by a declaration of the form:

type T **is delta** D **range** L .. R;

possibly with a small specification:

for T'SMALL **use** S;

is implicitly derived from a predefined fixed point type. The Compiler automatically selects the predefined fixed point type whose small and delta are the same as the small and delta of T and whose range is the shortest that includes the values L and R.

Encoding of fixed point values

In the program generated by the Compiler, a safe value V of a fixed point subtype F is represented as the integer:

V / F'BASE'SMALL

Fixed point subtypes

Minimum size: The minimum size of a fixed point subtype is the minimum number of binary digits that is necessary for representing the values of the range of the subtype using the small of the base type (that is to say, in an unbiased form which includes a sign bit only if the range of the subtype includes negative values).

For a static subtype, if it has a null range its minimum size is 1. Otherwise, s and S being the bounds of the subtype, if i and I are the integer representations of m and M, the smallest and the greatest model numbers of the base type such that s < m and M < S, then the minimum size L is determined as follows. For i >= 0, L is the smallest positive integer such that $I <= 2^L-1$. For i < 0, L is the smallest positive integer such that $-2^{L-1} <= i$ and $I <= 2^{L-1}-1$.

For example:

> **type** F **is delta** 2.0 **range** 0.0 .. 500.0;
> -- The minimum size of F is 8 bits.
>
> **subtype** S **is** F **delta** 16.0 **range** 0.0 .. 250.0;
> -- The minimum size of S is 7 bits.
>
> **subtype** D **is** S **range** X .. Y;
> -- Assuming that X and Y are not static, the minimum size of D is 7 bits
> -- (the same as the minimum size of its type mark S).

Size: The sizes of the sets of predefined fixed point types FIXED and LONG_FIXED are 16 and 32 bits respectively.

When no size specification is applied to a fixed point type or to its first named subtype, its size and the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly.

For example:

> **type** F **is delta** 0.01 **range** 0.0 .. 2.0;
> -- F is derived from a 16 bit predefined fixed type, its size is 16 bits.
>
> **type** L **is delta** 0.01 **range** 0.0 .. 300.0;
> -- L is derived from a 32 bit predefined fixed type, its size is 32 bits.
>
> **type** N **is new** L **range** 0.0 .. 2.0;
> -- N is indirectly derived from a 32 bit predefined fixed type, its size is 32 bits.

When a size specification is applied to a fixed point type, this fixed point type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies.

For example:

```
type F is delta 0.01 range 0.0 .. 2.0;
for F'SIZE use 32;
-- F is derived from a 16 bit predefined fixed type, but its size is 32 bits
-- because of the size specification.

type L is delta 0.01 range 0.0 .. 300.0;
for F'SIZE use 16;
-- F is derived from a 32 bit predefined fixed type, but its size is 16 bits
-- because of the size specification.
-- The size specification is legal since the range contains no negative values
-- and therefore no sign bit is required.

type N is new F range 0.8 .. 1.0;
-- N is indirectly derived from a 16 bit predefined fixed type, but its size is
-- 32 bits because N inherits the size specification of F.
```

The Alsys Compiler implements size specifications. Nevertheless, as fixed point objects are represented using machine integers, the specified length cannot be greater than 32 bits.

Object size: Provided its size is not constrained by a record component clause or a pragma PACK, an object of a fixed point type has the same size as its subtype.

Alignment: A fixed point subtype is byte aligned if its size is less than or equal to 8 bits, halfword aligned if the size of the subtype is less than or equal to 16 bits and word aligned otherwise.

Object address: Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a fixed point subtype is a multiple of the alignment of the corresponding subtype.

4.5 Access Types

Collection Size

When no specification of collection size applies to an access type, no storage space is reserved for its collection, and the value of the attribute STORAGE_SIZE is then 0.

As described in [13.2], a specification of collection size can be provided in order to reserve storage space for the collection of an access type. The Alsys Compiler fully implements this kind of specification.

Encoding of access values

Access values are machine addresses represented as 32 bit values. The implementation uses the top (most significant) bit of such a 32 bit value to pass additional information to the Ada Run-Time Executive.

Access subtypes

Minimum size: The minimum size of an access subtype is 32 bits.

Size: The size of an access subtype is 32 bits, the same as its minimum size.

The only size that can be specified for an access type using a size specification is its usual size (32 bits).

Object size: An object of an access subtype has the same size as its subtype, thus an object of an access subtype is always 32 bits long.

Alignment: An access subtype is always word aligned.

Object address: Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an access subtype is always on a word boundary, since its subtype is word aligned.

4.6 Task Types

Storage for a task activation

When no length clause is used to specify the storage space to be reserved for a task activation, the storage space indicated at bind time is used for this activation.

As described in [13.2], a length clause can be used to specify the storage space for the activation of each of the tasks of a given type. In this case the value indicated at bind time is ignored for this task type, and the length clause is obeyed.

It is not allowed to apply such a length clause to a derived type. The same storage space is reserved for the activation of a task of a derived type as for the activation of a task of the parent type.

Encoding of task values

Task values are machine addresses.

Task subtypes

Minimum size: The minimum size of a task subtype is 32 bits.

Size: The size of a task subtype is 32 bits, the same as its minimum size.

The only size that can be specified for a task type using a size specification is its usual size (32 bits).

Object size: An object of a task subtype has the same size as its subtype. Thus an object of a task subtype is always 32 bits long.

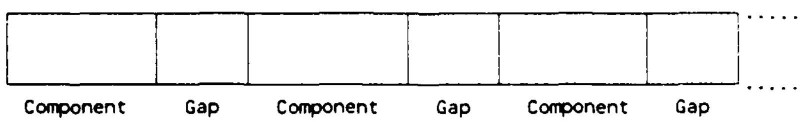
Alignment: A task subtype is always word aligned.

Object address: Provided its alignment is not constrained by a record representation clause, the address of an object of a task subtype is always on a word boundary, since its subtype is word aligned.

4.7 Array Types

Layout of an array

Each array is allocated in a contiguous area of storage units. All the components have the same size. A gap may exist between two consecutive components (and after the last one). All the gaps have the same size.



Components

If the array is not packed, the size of the components is the size of the subtype of the components.

For example:

```
type A is array (1 .. 8) of BOOLEAN;
-- The size of the components of A is the size of the type BOOLEAN: 8 bits.

type DECIMAL_DIGIT is range 0 .. 9;
for DECIMAL_DIGIT'SIZE use 4;
type BINARY_CODED_DECIMAL is
      array (INTEGER range < > ) of DECIMAL_DIGIT;
-- The size of the type DECIMAL_DIGIT is 4 bits. Thus in an array of
-- type BINARY_CODED_DECIMAL each component will be represented in
-- 4 bits as in the usual BCD representation.
```

If the array is packed and its components are neither records nor arrays, the size of the components is the minimum size of the subtype of the components.

For example:

```
type A is array (1 .. 8) of BOOLEAN;
pragma PACK(A);
-- The size of the components of A is the minimum size of the type BOOLEAN:
-- 1 bit.

type DECIMAL_DIGIT is range 0 .. 9;
type BINARY_CODED_DECIMAL is
      array (INTEGER range < > ) of DECIMAL_DIGIT;
pragma PACK(BINARY_CODED_DECIMAL);
-- The size of the type DECIMAL_DIGIT is 16 bits, but, as
-- BINARY_CODED_DECIMAL is packed, each component of an array of this
-- type will be represented in 4 bits as in the usual BCD representation.
```

Packing the array has no effect on the size of the components when the components are records or arrays.

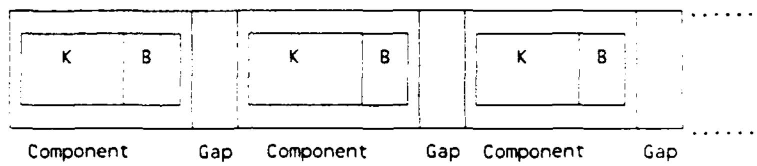
Gaps

If the components are records or arrays, no size specification applies to the subtype of the components and the array is not packed, then the Compiler may choose a representation with a gap after each component; the aim of the insertion of such gaps is to optimize access to the array components and to their subcomponents. The size of the gap is chosen so that the relative displacement of consecutive components is a multiple

of the alignment of the subtype of the components. This strategy allows each component and subcomponent to have an address consistent with the alignment of its subtype

For example:

```
type R is
    record
        K : INTEGER; -- INTEGER is word aligned.
        B : BOOLEAN; -- BOOLEAN is byte aligned.
    end record;
    -- Record type R is word aligned. Its size is 40 bits.

type A is array (1 .. 10) of R;
-- A gap of three bytes is inserted after each component in order to respect the
-- alignment of type R. The size of an array of type A will be 640 bits.
```

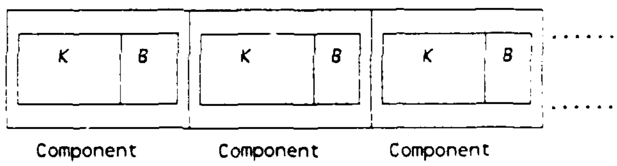


Array of type A: each subcomponent K has a word offset.

If a size specification applies to the subtype of the components or if the array is packed, no gaps are inserted.

For example:

```
type R is
      record
            K : INTEGER;
            B : BOOLEAN;
      end record;

type A is array (1 .. 10) of R;
pragma PACK(A);
-- There is no gap in an array of type A because A is packed.
-- The size of an object of type A will be 400 bits.

type NR is new R;
for NR'SIZE use 40;

type B is array (1 .. 10) of NR;
-- There is no gap in an array of type B because NR has a size specification.
-- The size of an object of type B will be 400 bits.
```



Array of type A or B: a subcomponent K can have any byte offset.

Array subtypes

Size: The size of an array subtype is obtained by multiplying the number of its components by the sum of the size of the components and the size of the gaps (if any). If the subtype is unconstrained, the maximum number of components is considered.

The size of an array subtype cannot be computed at compile time

- if it has non-static constraints or is an unconstrained array type with non-static index subtypes (because the number of components can then only be determined at run time).

- if the components are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static (because the size of the components and the size of the gaps can then only be determined at run time).

As has been indicated above, the effect of a pragma PACK on an array type is to suppress the gaps and to reduce the size of the components. The consequence of packing an array type is thus to reduce its size.

If the components of an array are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static, the Compiler ignores any pragma PACK applied to the array type but issues a warning message. Apart from this limitation, array packing is fully implemented by the Alsys Compiler.

The only size that can be specified for an array type or first named subtype using a size specification is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of an array is as expected by the application.

Object size: The size of an object of an array subtype is always equal to the size of the subtype of the object.

Alignment: If no pragma PACK applies to an array subtype and no size specification applies to its components, the array subtype has the same alignment as the subtype of its components.

If a pragma PACK applies to an array subtype or if a size specification applies to its components (so that there are no gaps), the alignment of the array subtype is the lesser of the alignment of the subtype of its components and the relative displacement of the components.

Object address: Provided its alignment is not constrained by a record representation clause, the address of an object of an array subtype is a multiple of the alignment of the corresponding subtype.

4.8 Record Types

Layout of a record

Each record is allocated in a contiguous area of storage units. The size of a record component depends on its type. Gaps may exist between some components.

The positions and the sizes of the components of a record type object can be controlled using a record representation clause as described in [13.4]. In the Alsys implementation

for IBM 370 machines there is no restriction on the position that can be specified for a component of a record. Bits within a storage unit are numbered from 0 to 7, with the most-significant bit numbered 0. The range of bits specified in a component clause may extend into following storage units. If a component is not a record or an array, its size can be any size from the minimum size to the size of its subtype. If a component is a record or an array, its size must be the size of its subtype:

```
type ACCESS_KEY is range 0..15;
-- The size of ACCESS_KEY is 16 bits, the minimum size is 4 bits

type CONDITIONS is (ZERO, LESS_THAN, GREATER_THAN, OVERFLOW);
-- The size of CONDITIONS is 8 bits, the minimum size is 2 bits

type PROG_EXCEPTION is (FIX_OVFL, DEC_OVFL, EXP_UNDFL, SIGNIF);
type PROG_MASK is array (PROG_EXCEPTION) of BOOLEAN;
pragma PACK (PROG_MASK);
-- The size of PROG_MASK is 4 bits

type ADDRESS is range 0..2**24-1;
for ADDRESS'SIZE use 24;
-- ADDRESS represents a 24 bit memory address

type PSW is
    record
            PER_MASK            : BOOLEAN;
            DAT_MODE            : BOOLEAN;
            IO_MASK             : BOOLEAN;
            EXTERNAL_MASK       : BOOLEAN;
            PSW_KEY             : ACCESS_KEY;
            EC_MODE             : BOOLEAN;
            MACHINE_CHECK       : BOOLEAN
            WAIT_STATE          : BOOLEAN;
            PROBLEM_STATE       : BOOLEAN;
            ADDRESS_SPACE       : BOOLEAN;
            CONDITION_CODE      : CONDITIONS;
            PROGRAM_MASK        : PROG_MASK;
            INSTR_ADDRESS       : ADDRESS;
    end record;
-- This type can be used to map the program status word of the IBM 370
```

```
for PSW use
    record at mod 8;
        PER_MASK           at 0    range 1..1;
        DAT_MODE           at 0    range 5..5;
        IO_MASK            at 0    range 6..6;
        EXTERNAL_MASK      at 0    range 7..7;
        PSW_KEY            at 1    range 0..3;
        EC_MODE            at 1    range 4..4;
        MACHINE_CHECK      at 1    range 5..5;
        WAIT_STATE         at 1    range 6..6;
        PROBLEM_STATE      at 1    range 7..7;
        ADDRESS_SPACE      at 2    range 0..0;
        CONDITION_CODE     at 2    range 2..3;
        PROGRAM_MASK       at 2    range 4..7;
        INSTR_ADDRESS      at 5    range 0..23;
    end record;
```

A record representation clause need not specify the position and the size for every component.

If no component clause applies to a component of a record, its size is the size of its subtype. Its position is chosen by the Compiler so as to optimize access to the components of the record: the offset of the component is chosen as a multiple of the alignment of the component subtype. Moreover, the Compiler chooses the position of the component so as to reduce the number of gaps and thus the size of the record objects.

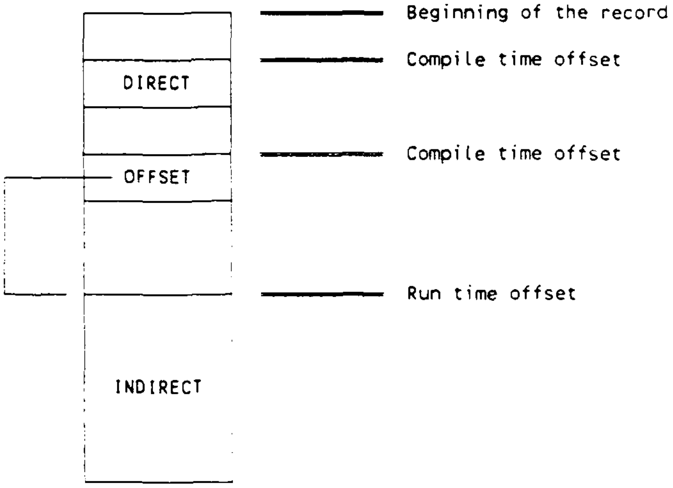
Because of these optimisations, there is no connection between the order of the components in a record type declaration and the positions chosen by the Compiler for the components in a record object.

Pragma PACK has no further effect on records. The Alsys Compiler always optimizes the layout of records as described above.

In the current version, it is not possible to apply a record representation clause to a derived type. The same storage representation is used for an object of a derived type as for an object of the parent type.

Indirect components

If the offset of a component cannot be computed at compile time, this offset is stored in the record objects at run time and used to access the component. Such a component is said to be indirect while other components are said to be direct:



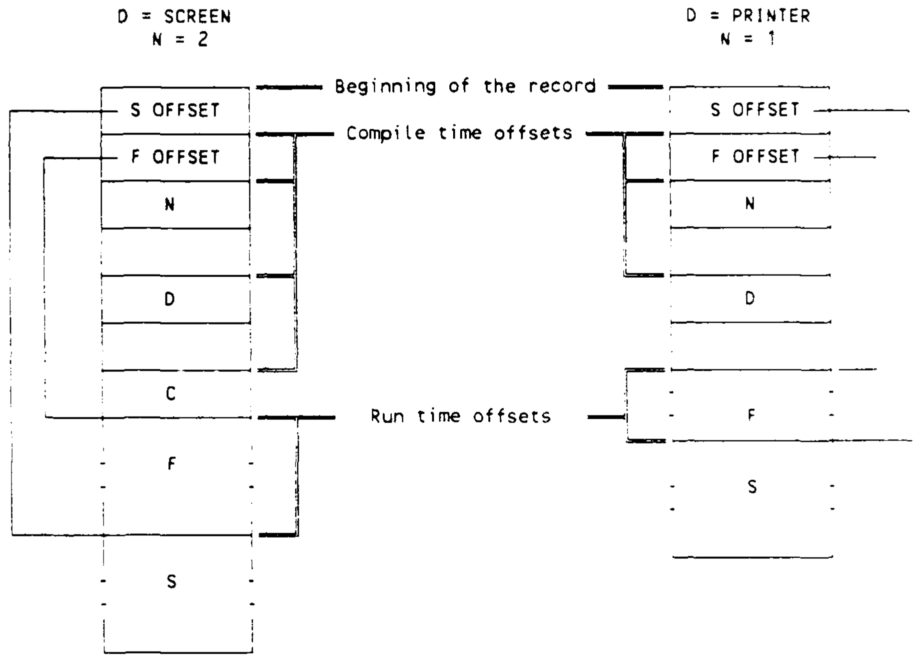
A direct and an indirect component

If a record component is a record or an array, the size of its subtype may be evaluated at run time and may even depend on the discriminants of the record. We will call these components dynamic components.

For example:

```
type DEVICE is (SCREEN, PRINTER);

type COLOR is (GREEN, RED, BLUE);

type SERIES is array (POSITIVE range < >) of INTEGER;

type GRAPH (L : NATURAL) is
    record
        X : SERIES(1 .. L); -- The size of X depends on L
        Y : SERIES(1 .. L); -- The size of Y depends on L
    end record;

Q : POSITIVE;

type PICTURE (N : NATURAL; D : DEVICE) is
    record
        F : GRAPH(N); -- The size of F depends on N
        S : GRAPH(Q); -- The size of S depends on Q
        case D is
            when SCREEN = >
              C : COLOR;
            when PRINTER = >
              null;
        end case;
    end record;
```

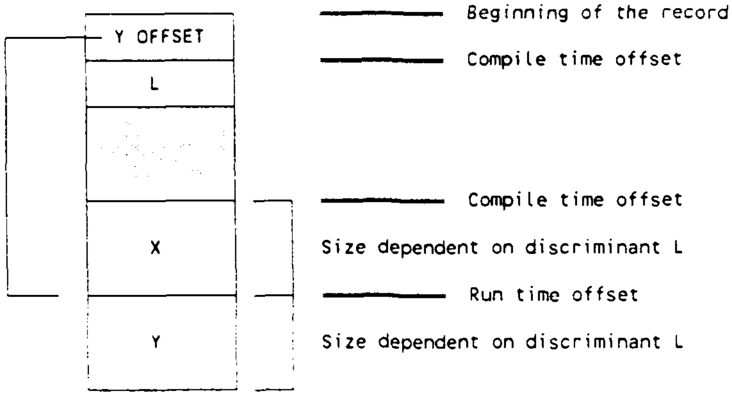
Any component placed after a dynamic component has an offset which cannot be evaluated at compile time and is thus indirect. In order to minimize the number of indirect components, the Compiler groups the dynamic components together and places them at the end of the record:



The record type PICTURE: F and S are placed at the end of the record

Thanks to this strategy, the only indirect components are dynamic components. But not all dynamic components are necessarily indirect: if there are dynamic components in a component list which is not followed by a variant part, then exactly one dynamic component of this list is a direct component because its offset can be computed at compilation time.

For example:



The record type GRAPH: the dynamic component X is a direct component.

The offset of an indirect component is always expressed in storage units.

The space reserved for the offset of an indirect component must be large enough to store the size of any value of the record type (the maximum potential offset). The Compiler evaluates an upper bound MS of this size and treats an offset as a component having an anonymous integer type whose range is 0 .. MS.

If C is the name of an indirect component, then the offset of this component can be denoted in a component clause by the implementation generated name C'OFFSET.

Implicit components

In some circumstances, access to an object of a record type or to its components involves computing information which only depends on the discriminant values. To avoid unnecessary recomputation, the Compiler stores this information in the record objects, updates it when the values of the discriminants are modified and uses it when the objects or their components are accessed. This information is stored in special components called implicit components.

An implicit component may contain information which is used when the record object or several of its components are accessed. In this case the component will be included in any record object (the implicit component is considered to be declared before any variant

part in the record type declaration). There can be two components of this kind; one is called RECORD_SIZE and the other VARIANT_INDEX.

On the other hand an implicit component may be used to access a given record component. In this case the implicit component exists whenever the record component exists (the implicit component is considered to be declared at the same place as the record component). Components of this kind are called ARRAY_DESCRIPTORs or RECORD_DESCRIPTORs.

RECORD_SIZE

This implicit component is created by the Compiler when the record type has a variant part and its discriminants are defaulted. It contains the size of the storage space necessary to store the current value of the record object (note that the storage effectively allocated for the record object may be more than this).

The value of a RECORD_SIZE component may denote a number of bits or a number of storage units. In general it denotes a number of storage units, but if any component clause specifies that a component of the record type has an offset or a size which cannot be expressed using storage units, then the value designates a number of bits.

The implicit component RECORD_SIZE must be large enough to store the maximum size of any value of the record type. The Compiler evaluates an upper bound MS of this size and then considers the implicit component as having an anonymous integer type whose range is 0 .. MS.

If R is the name of the record type, this implicit component can be denoted in a component clause by the implementation generated name R'RECORD_SIZE.

VARIANT_INDEX

This implicit component is created by the Compiler when the record type has a variant part. It indicates the set of components that are present in a record value. It is used when a discriminant check is to be done.

Component lists that do not contain a variant part are numbered. These numbers are the possible values of the implicit component VARIANT_INDEX.

For example:

```
type VEHICLE is (AIRCRAFT, ROCKET, BOAT, CAR);

type DESCRIPTION (KIND : VEHICLE := CAR) is
    record
        SPEED : INTEGER;
        case KIND is
            when AIRCRAFT | CAR =>
              WHEELS : INTEGER;
              case KIND is
                when AIRCRAFT =>    -- 1
                  WINGSPAN : INTEGER;
                when others =>      -- 2
                  null;
              end case;
            when BOAT =>            -- 3
              STEAM : BOOLEAN;
            when ROCKET =>          -- 4
              STAGES : INTEGER;
        end case;
    end record;
```

The value of the variant index indicates the set of components that are present in a record value:

Variant Index	Set
1	(KIND, SPEED, WHEELS, WINGSPAN)
2	(KIND, SPEED, WHEELS)
3	(KIND, SPEED, STEAM)
4	(KIND, SPEED, STAGES)

A comparison between the variant index of a record value and the bounds of an interval is enough to check that a given component is present in the value:

Component	Interval
KIND	--
SPEED	--
WHEELS	1 .. 2
WINGSPAN	1 .. 1
STEAM	3 .. 3
STAGES	4 .. 4

The implicit component VARIANT_INDEX must be large enough to store the number V of component lists that don't contain variant parts. The Compiler treats this implicit component as having an anonymous integer type whose range is 1 .. V.

If R is the name of the record type, this implicit component can be denoted in a component clause by the implementation generated name R'VARIANT_INDEX.

ARRAY_DESCRIPTOR

An implicit component of this kind is associated by the Compiler with each record component whose subtype is an anonymous array subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind ARRAY_DESCRIPTOR is not described in this documentation. Nevertheless, if a programmer is interested in specifying the location of a component of this kind using a component clause, he can obtain the size of the component using the ASSEMBLY parameter in the COMPILE command.

The Compiler treats an implicit component of the kind ARRAY_DESCRIPTOR as having an anonymous record type. If C is the name of the record component whose subtype is described by the array descriptor, then this implicit component can be denoted in a component clause by the implementation generated name C'ARRAY_DESCRIPTOR.

RECORD_DESCRIPTOR

An implicit component of this kind is associated by the Compiler with each record component whose subtype is an anonymous record subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind RECORD_DESCRIPTOR is not described in this documentation. Nevertheless, if a programmer is interested in specifying the location of a component of this kind using a component clause, he can obtain the size of the component using the ASSEMBLY parameter in the COMPILE command.

The Compiler treats an implicit component of the kind RECORD_DESCRIPTOR as having an anonymous record type. If C is the name of the record component whose subtype is described by the record descriptor, then this implicit component can be denoted in a component clause by the implementation generated name C'RECORD_DESCRIPTOR.

Suppression of implicit components

The Alsys implementation provides the capability of suppressing the implicit components RECORD_SIZE and/or VARIANT_INDEX from a record type. This can be done using an implementation defined pragma called IMPROVE. The syntax of this pragma is as follows:

pragma IMPROVE (TIME | SPACE , [ON =>] *simple_name*);

The first argument specifies whether TIME or SPACE is the primary criterion for the choice of the representation of the record type that is denoted by the second argument.

If TIME is specified, the Compiler inserts implicit components as described above. If on the other hand SPACE is specified, the Compiler only inserts a VARIANT_INDEX or a RECORD_SIZE component if this component appears in a record representation clause that applies to the record type. A record representation clause can thus be used to keep one implicit component while suppressing the other.

A pragma IMPROVE that applies to a given record type can occur anywhere that a representation clause is allowed for this type.

Record subtypes

Size: Unless a component clause specifies that a component of a record type has an offset or a size which cannot be expressed using storage units, the size of a record subtype is rounded up to a whole number of storage units.

The size of a constrained record subtype is obtained by adding the sizes of its components and the sizes of its gaps (if any). This size is not computed at compile time

- when the record subtype has non-static constraints,
- when a component is an array or a record and its size is not computed at compile time.

The size of an unconstrained record subtype is obtained by adding the sizes of the components and the sizes of the gaps (if any) of its largest variant. If the size of a component or of a gap cannot be evaluated exactly at compile time, an upper bound of this size is used by the Compiler to compute the subtype size.

The only size that can be specified for a record type or first named subtype using a size specification is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of a record is as expected by the application.

Object size: An object of a constrained record subtype has the same size as its subtype.

An object of an unconstrained record subtype has the same size as its subtype if this size is less than or equal to 8 Kbyte. If the size of the subtype is greater than this, the object has the size necessary to store its current value; storage space is allocated and released as the discriminants of the record change.

Alignment: When no record representation clause applies to its base type, a record subtype has the same alignment as the component with the highest alignment requirement.

When a record representation clause that does not contain an alignment clause applies to its base type, a record subtype has the same alignment as the component with the highest alignment requirement which has not been overridden by its component clause.

When a record representation clause that contains an alignment clause applies to its base type, a record subtype has an alignment that obeys the alignment clause.

Object address: Provided its alignment is not constrained by a representation clause, the address of an object of a record subtype is a multiple of the alignment of the corresponding subtype.

5 Conventions for Implementation-Generated Names

Special record components are introduced by the Compiler for certain record type definitions. Such record components are implementation-dependent; they are used by the Compiler to improve the quality of the generated code for certain operations on the record types. The existence of these components is established by the Compiler depending on implementation-dependent criteria. Attributes are defined for referring to them in record representation clauses. An error message is issued by the Compiler if the user refers to an implementation-dependent component that does not exist. If the implementation-dependent component exists, the Compiler checks that the storage

location specified in the component clause is compatible with the treatment of this component and the storage locations of other components. An error message is issued if this check fails.

There are four such attributes:

T'RECORD_SIZE	For a prefix T that denotes a record type. This attribute refers to the record component introduced by the Compiler in a record to store the size of the record object. This component exists for objects of a record type with defaulted discriminants when the sizes of the record objects depend on the values of the discriminants.
T'VARIANT_INDEX	For a prefix T that denotes a record type. This attribute refers to the record component introduced by the Compiler in a record to assist in the efficient implementation of discriminant checks. This component exists for objects of a record type with variant type.
C'ARRAY_DESCRIPTOR	For a prefix C that denotes a record component of an array type whose component subtype definition depends on discriminants. This attribute refers to the record component introduced by the Compiler in a record to store information on subtypes of components that depend on discriminants.
C'RECORD_DESCRIPTOR	For a prefix C that denotes a record component of a record type whose component subtype definition depends on discriminants. This attribute refers to the record component introduced by the Compiler in a record to store information on subtypes of components that depend on discriminants.

6 Address Clauses

6.1 Address Clauses for Objects

An address clause can be used to specify an address for an object as described in [13.5]. When such a clause applies to an object no storage is allocated for it in the program generated by the Compiler. The program accesses the object using the address specified in the clause.

An address clause is not allowed for task objects, nor for unconstrained records whose maximum possible size is greater than 8 Kbytes.

6.2 Address Clauses for Program Units

Address clauses for program units are not implemented.

6.3 Address Clauses for Entries

Address clauses for entries are not implemented.

7 Restrictions on Unchecked Conversions

Unconstrained arrays are not allowed as target types.

Unconstrained record types without defaulted discriminants are not allowed as target types.

If the source and the target types are each scalar or access types, the sizes of the objects of the source and target types must be equal.. If a composite type is used either as the source type or as the target type this restriction on the size does not apply.

If the source and the target types are each of scalar or access type or if they are both of composite type, the effect of the function is to return the operand.

In other cases the effect of unchecked conversion can be considered as a copy:

- if an unchecked conversion is achieved of a scalar or access source type to a composite target type, the result of the function is a copy of the source operand: the result has the size of the source.
- if an unchecked conversion is achieved of a composite source type to a scalar or access target type, the result of the function is a copy of the source operand: the result has the size of the target.

8 Input-Output Packages

The predefined input-output packages SEQUENTIAL_IO [14.2.3], DIRECT_IO [14.2.5], TEXT_IO [14.3.10] and IO_EXCEPTIONS [14.5] are implemented as described in the Language Reference Manual.

The package LOW_LEVEL_IO [14.6], which is concerned with low-level machine-dependent input-output, is not implemented.

8.1 NAME Parameter

The NAME parameter supplied to the Ada procedures CREATE or OPEN [14.2.1] must be a string which defines a legal path name under AIX.

8.2 FORM Parameter

The FORM parameter comprises a set of attributes formulated according to the lexical rules of [2], separated by commas. The FORM parameter may be given as a null string except when DIRECT_IO is instantiated with an unconstrained type; in this case the record size attribute must be provided. Attributes are comma-separated; blanks may be inserted between lexical elements as desired. In the descriptions below the meanings of *natural, positive*, etc., are as in Ada; attribute keywords (represented in upper case) are identifiers [2.3] and as such may be specified without regard to case.

USE_ERROR is raised if the FORM parameter does not conform to these rules.

The attributes are as follows:

8.2.1 File Protection

These attributes are only meaningful for a call to the CREATE procedure.

File protection involves two independent classifications. The first classification is related to *who* may access the file and is specified by the keywords:

OWNER	Only the owner of the directory may access this file.
GROUP	Only the members of a predefined group of users may access this file.

WORLD Any user may access this file.

For each type of user who may access a file there are various access *rights*, and this forms the basis for the second classification. In general, there are four types of access right, specified by the qualifiers:

READ The user may read from the external file.

WRITE The user may write to the external file.

EXECUTE The user may execute programs stored in the external file.

NONE The user has no access rights to the external file. (This access right negates any prior privileges.)

More than one access right may be relevant for a particular file, in which case the qualifiers are linked with underscores (_).

For example, suppose that the WORLD may execute a program in an external file, but only the OWNER may modify the file.

```
WORLD => EXECUTE, OWNER => READ_WRITE_EXECUTE,
```

Repetition of the same qualifier within the attributes is illegal:

```
WORLD =>EXECUTE_EXECUTE,                    -- NOT legal
```

but repetition of the entire attribute is allowed:

```
WORLD => EXECUTE, WORLD => EXECUTE,         -- Legal
```

8.2.2 File Sharing

An external file can be shared, which means associated simultaneously with several logical file objects created by the OPEN and CREATE procedures.

The file sharing attribute may restrict or suppress this capability by specifying one of the following access modes:

NOT_SHARED	Exclusive access - no other logical file may be associated with the external file
SHARED => READERS	Only logical files opened with mode IN are allowed
SHARED => SINGLE_WRITER	Only logical files opened with mode IN and at most one with mode INOUT or OUT are allowed
SHARED => ANY	No restriction

The exception USE_ERROR is raised if, for an external file already associated with an Ada file object:

- a further OPEN or CREATE specifies a file sharing attribute different from the current one
- a further OPEN, CREATE or RESET violates the conditions imposed by the current file sharing attribute.

The restrictions imposed by the file sharing attribute disappear when the last logical file object linked to the external file is closed.

The file sharing attribute provides control over multiple accesses within the program to a given external file.

This control does not extend to the whole system.

The default value for the file sharing attribute is SHARED => ANY

8.2.3 File Structure

Text Files

There is no FORM parameter to define the structure of text files.

A text file consists of a sequence of bytes holding the ASCII codes of characters.

The representation of Ada-terminators depends on the file's mode (IN or OUT) and whether it is associated with a terminal device or a disk file:

- Disk files

end of line:	ASCII.LF
end of page:	ASCII.LF ASCII.FF
end of file:	ASCII.LF

- Terminal device with mode IN

end of line:	ASCII.LF
end of page:	ASCII.FF
end of file:	ASCII.EOT

- Terminal device with mode OUT

end of line:	ASCII.LF
end of page:	ASCII.LF ASCII.FF
end of file:	ASCII.LF ASCII.FF

Binary Files

Two FORM attributes, RECORD_SIZE and RECORD_UNIT, control the structure of binary files.

A binary file can be viewed as a sequence (sequential access) or a set (direct access) of consecutive RECORDS.

The structure of such a record is:

    [HEADER] OBJECT [UNUSED_PART]

and it is formed from up to three items:

- an OBJECT with the exact binary representation of the Ada object in the executable program, possibly including an object descriptor
- a HEADER consisting of two fields (each of 32 bits):
  - the length of the object, in bytes when the object is a record and in bits when the object is an array
  - the length of the descriptor in bytes
- an UNUSED_PART of variable size to permit full control of the record's size

The HEADER is implemented only if the actual parameter of the instantiation of the IO package is unconstrained.

The file structure attributes take the form:

RECORD_SIZE => size_in_bytes
RECORD_UNIT => size_in_bytes

Their meaning depends on the object's type (constrained or not) and the file access mode (sequential or direct access):

a) If the object's type is constrained:

- The RECORD_UNIT attribute is illegal
- If the RECORD_SIZE attribute is omitted, no UNUSED_PART will be implemented: the default RECORD_SIZE is the object's size
- If present, the RECORD_SIZE attribute must specify a record size greater than or equal to the object's size, otherwise the exception USE_ERROR will be raised

b) If the object's type is unconstrained and the file access mode is direct:

- The RECORD_UNIT attribute is illegal
- The RECORD_SIZE attribute has no default value, and if it is not specified, a USE_ERROR will be raised
- An attempt to input or output an object larger than the given RECORD_SIZE will raise the exception DATA_ERROR

c) If the object's type is unconstrained and the file access mode is sequential:

- The RECORD_SIZE attribute is illegal
- The default value of the RECORD_UNIT attribute is 1 (byte)
- The record size will be the smallest multiple of the specified (or default) RECORD_UNIT that holds the object and its header. This is the only case where records of a file may have different sizes.

8.2.4 Buffering

The buffer size can be specified by the attribute

BUFFER_SIZE => size_in_bytes

The default value for BUFFER_SIZE is 0 (which means no buffering) for terminal devices; it is 1 block for disk files.

8.2.5 Appending

Only to be used with the procedure OPEN, the format of this attribute is simply

APPEND

and it means that any output will be placed at the end of the named external file.

In normal circumstances, when an external file is opened, an index is set which points to the beginning of the file. If the APPEND attribute is present for a sequential or for a text file, then data transfer will commence at the end of the file. For a direct access file, the value of the index is set to one more than the number of records in the external file.

This attribute is not applicable to terminal devices.

8.2.6 Blocking

This attribute has two alternative forms:

BLOCKING ,

or

NON_BLOCKING ,

This attribute specifies the IO system behavior desired at any moment that a request for data transfer cannot be fulfilled. The stoppage may be due, for example, to the unavailability of data, or to the unavailability of the external file device.

NON_BLOCKING

If this attribute is set, then the task that ordered the data transfer is suspended - meaning that other tasks can execute. The suspended task is kept in a 'ready' state, together with other tasks in a ready state at the same priority level (that is, it is rescheduled).

When the suspended task is next scheduled, the data transfer request is reactivated. If ready, the transfer is activated, otherwise the rescheduling is repeated. Control returns to the user program after completion of the data transfer.

BLOCKING

In this case the task waits until the data transfer is complete, and all other tasks are suspended (or 'blocked'). The system is busy waiting.

The default for this attribute depends on the actual program: it is BLOCKING for programs without task declarations and NON_BLOCKING for a program containing tasks.

8.2.7 Terminal Input

This attribute takes one of two alternative forms:

```
TERMINAL_INPUT => LINES,
TERMINAL_INPUT => CHARACTERS.
```

Terminal input is normally processed in units of a line at a time, where a line is delimited by a special character. A process attempting to read from the terminal as an external file will be suspended until a complete line has been typed. At that time, the outstanding read call (and possibly also later calls) will be satisfied.

The first option specifies line-at-a-time data transfer, which is the default case.

The second option means that data transfer is character by character, and so a complete line does not have to be entered before the read request can be satisfied. For this option the BUFFER_SIZE (see section 8.2.4) must be zero.

The TERMINAL_INPUT attribute is only applicable to terminal devices.

8.3 STANDARD_INPUT and STANDARD_OUTPUT

The Ada internal files STANDARD_INPUT and STANDARD_OUTPUT are associated with the external streams *stdin* and *stdout*, respectively. By default under AIX the *stdin* and *stdout* streams are defined to be the terminal, but the user may redefine them by using the IO redirection symbols (<, > and >>). The < (less than) symbol can be used to take input from a file. The > (greater than) symbol can be used to send output to a file, overwriting any original contents. The >> symbol can be used to append output to a file.

8.4 USE_ERROR

The following conditions will cause USE_ERROR to be raised:

- Specifying a FORM parameter whose syntax does not conform to the rules given above.
- Specifying the RECORD_SIZE FORM parameter attribute to have a value of zero, or failing to specify RECORD_SIZE for instantiations of DIRECT_IO for unconstrained types.
- Specifying a RECORD_SIZE FORM parameter attribute to have a value less than that required to hold the element for instantiations of DIRECT_IO and SEQUENTIAL_IO for constrained types.
- Violating the file sharing rules stated above.
- Errors detected whilst reading or writing (e.g. writing to a file on a read-only disk).

8.5 Text Terminators

Line terminators [14.3] are implemented using the ASCII.NL character 0A (hexadecimal) and are implied by the end of physical record.

Page terminators [14.3] are implemented using the ASCII.NP character 0C (hexadecimal).

File terminators [14.3] are implemented using the ASCII.EOT character 04 (hexadecimal) and are implied by the end of physical file.

The user should avoid the explicit output of the character ASCII.NP [C], as this will not cause a page break to be emitted. If the user explicitly outputs the character ASCII.LF, this is treated as a call of NEW_LINE [14.3.4].

9 Characteristics of Numeric Types

9.1 Integer Types

The ranges of values for integer types declared in package STANDARD are as follows:

SHORT_SHORT_INTEGER	-128 .. 127	-- -2**7 .. 2**7 - 1
SHORT_INTEGER	-32768 .. 32767	-- -2**15 .. 2**15 - 1
INTEGER	-2147483648 .. 2147483647	-- -2**31 .. 2**31 - 1

For the packages DIRECT_IO and TEXT_IO, the ranges of values for types COUNT and POSITIVE_COUNT are as follows:

COUNT	0 .. 2147483647	-- 0 .. 2**31 - 1
POSITIVE_COUNT	1 .. 2147483647	-- 1 .. 2**31 - 1

For the package TEXT_IO, the range of values for the type FIELD is as follows:

FIELD	0 .. 255	-- 0 .. 2**8 - 1

9.2 Floating Point Type Attributes

SHORT_FLOAT

		Approximate value
DIGITS	6	
MANTISSA	21	
EMAX	84	
EPSILON	2.0 ** -20	9.54E-07
SMALL	2.0 ** -85	2.58E-26
LARGE	2.0 ** 84 * (1.0 - 2.0 ** -21)	1.93E+25
SAFE_EMAX	252	
SAFE_SMALL	2.0 ** -253	6.91E-77
SAFE_LARGE	2.0 ** 252 * (1.0 - 2.0 ** -21)	7.24E+75
FIRST	-2.0 ** 252 * (1.0 - 2.0 ** -24)	-7.24E+75
LAST	2.0 ** 252 * (1.0 - 2.0 ** -24)	7.24E+75
MACHINE_RADIX	16	
MACHINE_MANTISSA	6	
MACHINE_EMAX	63	
MACHINE_EMIN	-64	
MACHINE_ROUNDS	FALSE	
MACHINE_OVERFLOWS	TRUE	
SIZE	32	

FLOAT

		Approximate value
DIGITS	15	
MANTISSA	51	
EMAX	204	
EPSILON	2.0 ** -50	8.88E-16
SMALL	2.0 ** -205	1.94E-62
LARGE	2.0 ** 204 * (1.0 - 2.0 ** -51)	2.57E+61
SAFE_EMAX	252	
SAFE_SMALL	2.0 ** -253	6.91E-77
SAFE_LARGE	2.0 ** 252 * (1.0 - 2.0 ** -51)	7.24E+75
FIRST	-2.0 ** 252 * (1.0 - 2.0 ** -56)	-7.24E+75
LAST	2.0 ** 252 * (1.0 - 2.0 ** -56)	7.24E+75
MACHINE_RADIX	16	
MACHINE_MANTISSA	14	
MACHINE_EMAX	63	
MACHINE_EMIN	-64	
MACHINE_ROUNDS	FALSE	
MACHINE_OVERFLOWS	TRUE	
SIZE	64	

LONG_FLOAT

		Approximate value
DIGITS	18	
MANTISSA	61	
EMAX	244	
EPSILON	2.0 ** -60	8.67E-19
SMALL	2.0 ** -245	1.77E-74
LARGE	2.0 ** 244 * (1.0 - 2.0 ** -61)	2.83E+73
SAFE_EMAX	252	
SAFE_SMALL	2.0 ** -253	6.91E-77
SAFE_LARGE	2.0 ** 252 * (1.0 - 2.0 ** -61)	7.24E+75
FIRST	-2.0 ** 252 * (1.0 - 2.0 ** -112)	-7.24E+75
LAST	2.0 ** 252 * (1.0 - 2.0 ** -112)	7.24E+75
MACHINE_RADIX	16	
MACHINE_MANTISSA	28	
MACHINE_EMAX	63	
MACHINE_EMIN	-64	
MACHINE_ROUNDS	FALSE	
MACHINE_OVERFLOWS	TRUE	
SIZE	128	

9.3 Attributes of Type DURATION

DURATION'DELTA	2.0 ** -14	
DURATION'SMALL	2.0 ** -14	
DURATION'LARGE	131072.0DURATION'FIRST	-131072.0
DURATION'LAST	131071.0	

10 Other Implementation-Dependent Characteristics

10.1 Characteristics of the Heap

All objects created by allocators go into the program heap. In addition, portions of the Ada Run-Time Executive's representation of task objects, including the task stacks, are allocated in the program heap.

All objects on the heap belonging to a given collection have their storage reclaimed on exit from the innermost block statement, subprogram body or task body that encloses the access type declaration associated with the collection. For access types declared at the library level, this deallocation occurs only on completion of the main program.

There is no further automatic storage reclaimation performed, i.e. in effect, all access types are deemed to be controlled [4.8]. The explicit deallocation of the object designated by an access value can be achieved by calling an appropriate instantiation of the generic procedure UNCHECKED_DEALLOCATION.

Space for the heap is initially claimed from the system on program start up and additional space may be claimed as required when the initial allocation is exhausted. The size of both the initial allocation and the size of the individual increments claimed from the system may be controlled by the Binder options SIZE and INCREMENT. Corresponding run-time options also exist.

On an extended architecture machine space allocated from the program heap may be above or below the 16 megabyte virtual storage line.

10.2 Characteristics of Tasks

The default initial task stack size is 16 Kbytes, but by using the Binder option TASK the size for all task stacks in a program may be set to any size from 4 Kbytes to 16 Mbytes. A corresponding run-time option also exists.

If a task stack becomes exhausted during execution, it is automatically extended using storage claimed from the heap. The TASK option specifies the minimum size of such an extension, i.e. the task stack is extended by the size actually required or by the value of the TASK option, whichever is the larger.

Timeslicing is implemented for task scheduling. The default time slice is 1000 milliseconds, but by using the Binder option SLICE the time slice may be set to any multiple of 10 milliseconds. A corresponding run-time option also exists. It is also possible to use this option to specify no timeslicing, i.e. tasks are scheduled only at explicit synchronisation points. Timeslicing is started only upon activation of the first task in the program, so the SLICE option has no effect for sequential programs.

Normal priority rules are followed for preemption, where PRIORITY values run in the range 1 .. 10. All tasks with "undefined" priority (no pragma PRIORITY) are considered to have a priority of 0.

The minimum timeable delay is 10 milliseconds.

The maximum number of active tasks is limited only by memory usage. Tasks release their storage allocation as soon as they have terminated.

The acceptor of a rendezvous executes the accept body code in its own stack. A rendezvous with an empty accept body (e.g. for synchronisation) need not cause a context switch.

The main program waits for completion of all tasks dependent on library packages before terminating. Such tasks may select a terminate alternative only after completion of the main program.

Abnormal completion of an aborted task takes place immediately, except when the abnormal task is the caller of an entry that is engaged in a rendezvous. Any such task becomes abnormally completed as soon as the rendezvous is completed.

If a global deadlock situation arises because every task (including the main program) is waiting for another task, the program is aborted and the state of all tasks is displayed.

10.3 Definition of a Main Program

A main program must be a non-generic, parameterless, library procedure.

10.4 Ordering of Compilation Units

The Alsys IBM 370 AIX Ada Compiler imposes no additional ordering constraints on compilations beyond those required by the language.

INDEX

NCC VSR ADDENDUM

This Addendum to the ACVC 1.11 VSR clarifies some items which are contained within the standard pro-forma Validation Summary Report as supplied by the Ada Maintenance Organisation (AMO).

In line with AJPO regulations the contents of the VSR have not been altered in order to keep consistency between the different AVF's.

The points raised in this addendum are being addressed by the AMO in future issues of the VSR.

1 Chapter 1 of the VSR does not indicate how 'inapplicable' tests are to be analysed. The analysis is undertaken as follows:

'Each inapplicable test is checked to ensure that this behaviour is consistent with the given reasons for its inapplicability'.

2 The Test counts for this VSR are as follows:

Total Tests Passed	3858
Total Tests Not Applicable	241
Total Tests Withdrawn	71
Total Tests	4170